

Tabla de contenidos

- 1. Ejecutables. Procesos. Servicios.**
- 2. Hilos**
- 3. Programación concurrente**
- 4. Programación paralela y distribuida**
- 5. Creación de procesos**
- 6. Comunicación entre procesos**
- 7. Gestión de procesos**
- 8. Comandos para la gestión de procesos en sistemas libres y propietarios**
- 9. Sincronización entre procesos**

1. Ejecutables. Procesos. Servicios.

Ejecutables

Un ejecutable es un archivo con la estructura necesaria para que el sistema operativo pueda poner en marcha el programa que hay dentro. En Windows, los ejecutables suelen ser archivos con la extensión .EXE.

Se pueden utilizar «desensambladores» para averiguar la secuencia de instrucciones que hay en un EXE. Incluso existen desensambladores en línea como <http://onlinedisassembler.com>

Sin embargo, Java genera ficheros .JAR o .CLASS. Estos ficheros *no son ejecutables* sino que son archivos que el intérprete de JAVA (el archivo `java.exe`) leerá y ejecutará.

El intérprete toma el programa y lo traduce a instrucciones del microprocesador en el que estemos, que puede ser x86 o un x64 o lo que sea. Ese proceso se hace «al instante» o JIT (Just-In-Time).

Un archivo .CLASS puede desensamblarse utilizando el comando `javap -c <archivo.class>`. Cuando se hace así, se obtiene un listado de «instrucciones» que no se corresponden con las instrucciones del microprocesador, sino con «instrucciones virtuales de Java». El intérprete Java (el archivo `java.exe`) traducirá en el momento del arranque dichas instrucciones virtuales Java a instrucciones reales del microprocesador.

Este último aspecto es el esgrimido por Java para defender que su ejecución puede ser más rápida que la de un EXE, ya que Java puede averiguar en qué microprocesador se está ejecutando y así generar el código más óptimo posible.

Un EXE puede que no contenga las instrucciones de los microprocesadores más modernos. Como todos son compatibles no es un gran problema, sin embargo, puede que no aprovechemos al 100% la capacidad de nuestro micro.

Procesos

Es un archivo que está en ejecución y bajo el control del sistema operativo. Un proceso puede atravesar diversas etapas en su «ciclo de vida». Los estados en los que puede estar son:

- En ejecución: está dentro del microprocesador.
- Pausado/detenido/en espera: el proceso tiene que seguir en ejecución pero en ese momento el S.O tomó la decisión de dejar paso a otro.
- Interrumpido: el proceso tiene que seguir en ejecución pero *el usuario* ha decidido interrumpir la ejecución.
- Existen otros estados pero ya son muy dependientes del sistema operativo concreto.

Servicios

Un servicio es un proceso que no muestra ninguna ventana ni gráfico en pantalla porque no está pensado para que el usuario lo maneje directamente.

Habitualmente, un servicio es un programa que atiende a otro programa.

2. Hilos

Un hilo es un concepto más avanzado que un proceso: al hablar de procesos cada uno tiene su propio espacio en memoria. Si abrimos 20 procesos cada uno de ellos consume 20x de memoria RAM. Un hilo es un proceso mucho más ligero, en el que el código y los datos se comparten de una forma distinta.

Un proceso no tiene acceso a los datos de otro procesos. Sin embargo un hilo sí accede a los datos de otro hilo. Esto complicará algunas cuestiones a la hora de programar.

3. Programación concurrente

La programación concurrente es la parte de la programación que se ocupa de crear programas que pueden tener varios procesos/hilos que colaboran para ejecutar un trabajo y aprovechar al máximo el rendimiento de sistemas multinúcleo. En el caso de la programación concurrente un solo ordenador puede ejecutar varias tareas a la vez (lo que supone que tiene 2 o más núcleos).

Por otro lado se denomina programación paralela a la capacidad de un núcleo de ejecutar dos o más tareas a la vez, normalmente repartiendo el tiempo de proceso entre las tareas.

4. Programación paralela y distribuida

Dentro de la programación concurrente tenemos la paralela y la distribuida:

- En general se denomina «programación paralela» a la creación de software que se ejecuta siempre en un solo ordenador (con varios núcleos o no)
- Se denomina «programación distribuida» a la creación de software que se ejecuta en ordenadores distintos y que se comunican a través de una red.

5. Creación de procesos

En Java es posible crear procesos utilizando algunas clases que el entorno ofrece para esta tarea. En este tema, veremos en profundidad la clase `ProcessBuilder`.

El ejemplo siguiente muestra como lanzar un proceso de Acrobat Reader:

```
public class LanzadorProcesos {  
    public void ejecutar(String ruta){  
  
        ProcessBuilder pb;  
        try {  
            pb = new ProcessBuilder(ruta);  
            pb.start();  
        } catch (Exception e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
  
    }  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        String ruta=  
            "C:\\Program Files (x86)\\Adobe\\Reader 11.0\\Reader\\AcroRd32.exe";  
        LanzadorProcesos lp=new LanzadorProcesos();  
        lp.ejecutar(ruta);  
        System.out.println("Finalizado");  
    }  
}
```

Supongamos que necesitamos crear un programa que aproveche al máximo el número de CPUs para realizar alguna tarea intensiva.
Supongamos que dicha tarea consiste en sumar números.

Enunciado: crear una clase Java que sea capaz de sumar todos los números comprendidos entre dos valores incluyendo ambos valores.

Para resolverlo crearemos una clase `Sumador` que tenga un método que acepte dos números `n1` y `n2` y que devuelva la suma de todo el intervalo.

Además, incluiremos un método `main` que ejecute la operación de suma tomando los números de la línea de comandos (es decir, se pasan como argumentos al `main`).

El código de dicha clase podría ser algo así:

```
public class Sumador {  
    public static int sumar(int n1, int n2){  
        int suma=0;  
        if (n1>n2){  
            int aux=n1;  
            n1=n2;  
            n2=aux;  
        }  
        for (int i=n1; i<=n2; i++){  
            suma=suma+i;  
        }  
        return suma;  
    }  
  
    public static void main(String[] args){  
        int n1=Integer.parseInt(args[0]);  
        int n2=Integer.parseInt(args[1]);  
        int suma=sumar(n1, n2);  
        System.out.println(suma);  
        System.out.flush();  
    }  
}
```

Para ejecutar este programa desde dentro del IDE es necesario indicar que deseamos enviar *argumentos* al programa.

Una vez hecha la prueba de la clase sumador, tendremos una clase que lanza procesos de esta forma:

```
import java.io.File;

public class Lanzador {
    public void lanzarSumador(Integer n1, Integer n2){
        String clase= System.getProperty("user.dir") + "\\src\\Sumador.java";
        ProcessBuilder pb;
        try {
            pb = new ProcessBuilder(
                "java",clase,
                n1.toString(),
                n2.toString());
            pb.start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args){
        Lanzador l=new Lanzador();
        l.lanzarSumador(1, 5);
        l.lanzarSumador(6, 10);
        System.out.println("Ok");
    }
}
```

6. Comunicación entre procesos

Las operaciones multiproceso pueden implicar que sea necesario comunicar información entre muchos procesos, lo que obliga a la necesidad de utilizar mecanismos específicos de comunicación que ofrecerá Java o a diseñar alguno separado que evite los problemas que puedan aparecer.

En el ejemplo, el segundo proceso suele sobreescribir el resultado del primero, así que modificaremos el código del lanzador para que cada proceso use su propio fichero de resultados.

```
import java.io.File;

public class Lanzador {
    public void lanzarSumador(Integer n1,
        Integer n2, String fichResultado){
        String clase= System.getProperty("user.dir") + "\\src\\Sumador.java";
        ProcessBuilder pb;
        try {
            pb = new ProcessBuilder(
                "java",clase,
                n1.toString(),
                n2.toString());

            pb.redirectError(new File("errores.txt"));
            pb.redirectOutput(new File(fichResultado));
            pb.start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args){
        Lanzador l=new Lanzador();
        l.lanzarSumador(1, 5, "result1.txt");
        l.lanzarSumador(6,10, "result2.txt");
        System.out.println("Ok");
    }
}
```

Cuando se lanza un programa desde un IDE puede que no ocurra lo mismo que cuando se lanza desde Windows. Los IDEs trabajan con unos directorios predefinidos y puede ser necesario indicar a nuestro programa cual es la ruta donde hay que buscar algo.

Usando el método `.directory(new File("c:\\dir\\"))` se puede indicar a Java donde está el archivo que se desea ejecutar.

7. Gestión de procesos

La gestión de procesos se realiza de dos formas **muy distintas** en función de los dos grandes sistemas operativos: Windows y Linux.

- En Windows toda la gestión de procesos se realiza desde el «Administrador de tareas» al cual se accede con Ctrl+Alt+Supr. Existen otros programas más sofisticados que proporcionan algo más de información sobre los procesos, como Process Explorer (antes conocido con el nombre de ProcessViewer).

8. Comandos para la gestión de procesos en sistemas libres y propietarios

En sistemas Windows, no existen apenas comandos para gestionar procesos. Puede obligarse al sistema operativo a arrancar la aplicación asociada a un archivo con el comando `START`. Es decir, si se ejecuta lo siguiente:

```
START documento.pdf
```

se abrirá el visor de archivos PDF el cual cargará automáticamente el fichero `documento.pdf`

En GNU/Linux se puede utilizar un terminal de consola para la gestión de procesos, lo que implica que no solo se pueden arrancar procesos si no tambien detenerlos, reanudarlos, terminarlos y modificar su prioridad de ejecución.

- Para arrancar un proceso, simplemente tenemos que escribir el nombre del comando correspondiente. Desde GNU/Linux se pueden controlar los servicios que se ejecutan con un comando llamado `service`. Por ejemplo, se puede usar `sudo service apache2 stop` para parar el servidor web y `sudo service apache2 start` para volver a ponerlo en marcha. También se puede reiniciar un servicio (tal vez para que relea un fichero de configuración que hemos cambiado) con `sudo service apache2 restart`.
- Se puede detener y/o terminar un proceso con el comando `kill`. Se puede usar este comando para **terminar un proceso** sin guardar nada usando `kill -SIGKILL <numpresc>` o `kill -9 <numpresc>`. Se puede pausar un proceso con `kill -SIGSTOP <numpresc>` y rearrancarlo con `kill -SIGCONT`
- Se puede enviar un proceso a segundo plano con comandos como `bg` o al arrancar el proceso escribir el nombre del comando terminado en `&`.
- Se puede devolver un proceso a primer plano con el comando `fg`.

Prioridades

En sistemas como GNU/Linux se puede modificar la prioridad con que se ejecuta un proceso. Esto implica dos posibilidades

- Si pensamos que un programa que necesitamos ejecutar es muy importante podemos darle más prioridad para que reciba «más turnos» del planificador.
- Y por el contrario, si pensamos que un programa no es muy necesario podemos quitarle prioridad y reservar «más turnos de planificador» para otros posibles procesos.

El comando `nice` permite indicar prioridades entre -20 y 19. El -20 implica que un proceso reciba la **máxima prioridad**, y el 19 supone asignar la **mínima prioridad**

En windows también se puede usar el [Process Explorer](#).

9. Sincronización entre procesos

Cuando se lanza más de un proceso de una misma sección de código no se sabe qué proceso ejecutará qué instrucción en un cierto momento, lo que es muy peligroso:

```
int i,j;
i=0;
if (i>=2){
    i=i+1;
    j=j+1
}
System.out.println("Ok");
i=i*2;
j=j-1;
```

Si dos o más procesos avanzan por esta sección de código es perfectamente posible que unas veces nuestro programa multiproceso se ejecute bien y otras no.

En todo programa multiproceso pueden encontrarse estas zonas de código «peligrosas» que deben protegerse especialmente utilizando ciertos mecanismos. El nombre global para todos los lenguajes es denominar a estos trozos «secciones críticas».

Mecanismos para controlar secciones críticas

Los mecanismos más típicos son los ofrecidos por UNIX/Windows:

- Semáforos.
- Colas de mensajes.
- Tuberías (pipes)
- Bloques de memoria compartida.

En realidad algunos de estos mecanismos se utilizan más para intercomunicar procesos, aunque para los programadores Java la forma de resolver el problema de la «sección crítica» es más simple.

En Java, si el programador piensa que un trozo de código es peligroso puede ponerle la palabra clave `synchronized` y la máquina virtual Java protege el código automáticamente.

```
/* La máquina virtual Java evitará que más de un proceso/hilo acceda a este método*/
synchronized
    public void actualizarPension(int nuevoValor){
        /*..trozo de código largo omitido*/
        this.pension=nuevoValor
    }

    /* Otro ejemplo, ahora no hemos protegido un método entero,
    sino solo un pequeño trozo de código.*/
    for (int i=0; i=i+1; i++){
        /* Código omitido*/
        synchronized {
            i=i*2;
            j=j+1;
        }
    }
```